RADICALLY OPEN SECURITY

NInet Security Evaluation Report

rPGP

V 1.0
Amsterdam, December 5th, 2024
Public

Document Properties

Client	rPGP
Title	NLnet Security Evaluation Report
Targets	Limited code review of rPGP project Fuzz testing of exposed rPGP attack surface Assist the rPGP project in reproducing, fixing and reporting of discovered issues Discuss potential improvements for documentation and testing Prepare new test code for adoption by rPGP project
Version	1.0
Pentester	Christian Reitter
Authors	Christian Reitter, Marcus Bointon
Reviewed by	Marcus Bointon
Approved by	Melanie Rieback

Version control

Version	Date	Author	Description
0.1	November 29th, 2024	Christian Reitter	Initial draft
0.2	November 30th, 2024	Marcus Bointon	Review
1.0	December 5th, 2024	Marcus Bointon	1.0

Contact

For more information about this document and its contents please contact Radically Open Security B.V.

Name	Melanie Rieback
Address	Science Park 608 1098 XH Amsterdam The Netherlands
Phone	+31 (0)20 2621 255
Email	info@radicallyopensecurity.com

Table of Contents

1 Executive Summary

1.1 Introduction

Between November 4, 2024 and November 29, 2024, Radically Open Security B.V. carried out a penetration test for rPGP and NLnet NGI Zero Core as part of the Support for OpenPGP v6 in `rPGP` project grant.

This report contains our findings as well as detailed explanations of exactly how ROS performed the penetration test.

1.2 Scope of work

The scope of the penetration test was limited to the following targets:

- Limited code review of `rPGP` project
- Fuzz testing of exposed `rPGP` attack surface
- Assist the `rPGP` project in reproducing, fixing and reporting of discovered issues
- Discuss potential improvements for documentation and testing
- Prepare new test code for adoption by `rPGP` project

The scoped services are broken down as follows:

- Pentesting: 3 days
- Reporting: 8 hours
- Communication, shared sessions and other tasks: 8 hours
- Time extension to analyze practical impact, perform retesting, assistance for disclosure: 4 days
- **Total effort: 9 days**

1.3 Project objectives

ROS will perform a code review of the rPGP library with rPGP in order to assess its security. To do so ROS will perform a code review on the newest public code revision of the Open Source project and guide rPGP in attempting to find vulnerabilities, exploiting and analyzing any such found to try and gain further understanding of the security impact. Where possible, ROS will assist with retesting of fixes and mitigations that become available during the engagement phase.

Due to the limited time available for review and the main scope arranged with rPGP, the focus of this evaluation is mainly on vulnerability classes which can be found via existing static analysis and dynamic analysis testing frameworks. Many other cryptographic problems such as non-constant time computation and other side channels are out of scope for this review and were not analyzed.

Reviewed rPGP version: commit `3a1bb5cb5ba88370a6966c7f5d4a48e2dd839001` on the `main` branch. This was the newest available code revision at the beginning of the review, and represents the code state shortly after the `v0.14.0` stable release.

The retest results apply to the `v0.14.2` stable release.

1.4 Timeline

The security audit took place between November 4, 2024 and November 29, 2024.

1.5 Staffing Note

The ROS pentester Christian Reitter and rPGP developer Heiko Schaefer have worked together on unpaid security research projects and security disclosures outside of this engagement. This was discussed within ROS and with rPGP before the beginning of the review, and was not seen as a conflict of interest. We still include it here as a note for transparency.

1.6 Results In A Nutshell

During this crystal-box penetration test we found 5 High, 1 Elevated, 2 Moderate and 6 Low-severity issues.

The findings relate to incorrect length handling, uncaught exceptions, reachable assertions, risky functions, integer overflow edge cases, missing memory sanitization, and uncontrolled resource use.

By exploiting these issues, an attacker might be able to crash the `rPGP` library and associated processes, cause very long processing times, affect the system stability via memory resource exhaustion, have increased chances of obtaining sensitive memory values via other vulnerabilities, and trigger unexpected edge cases in the cryptographic handling.

Two of the findings only apply to older `rPGP` versions and are no longer present on the primarily analyzed `rPGP` revision.

1.7 Public Advisories

rPGP created two public advisories with CVE identifiers for some of the discovered findings, and ROS assisted with this effort.

Advisory GHSA-9rmp-2568-59rv (CVE-2024-53856) covers potential program crashes via RPG-007 (page 14), RPG-015 (page 16), RPG-016 (page 17), RPG-017 (page 28), RPG-019 (page 19), RPG-020 (page 35), RPG-021 (page 37), and RPG-022 (page 20).

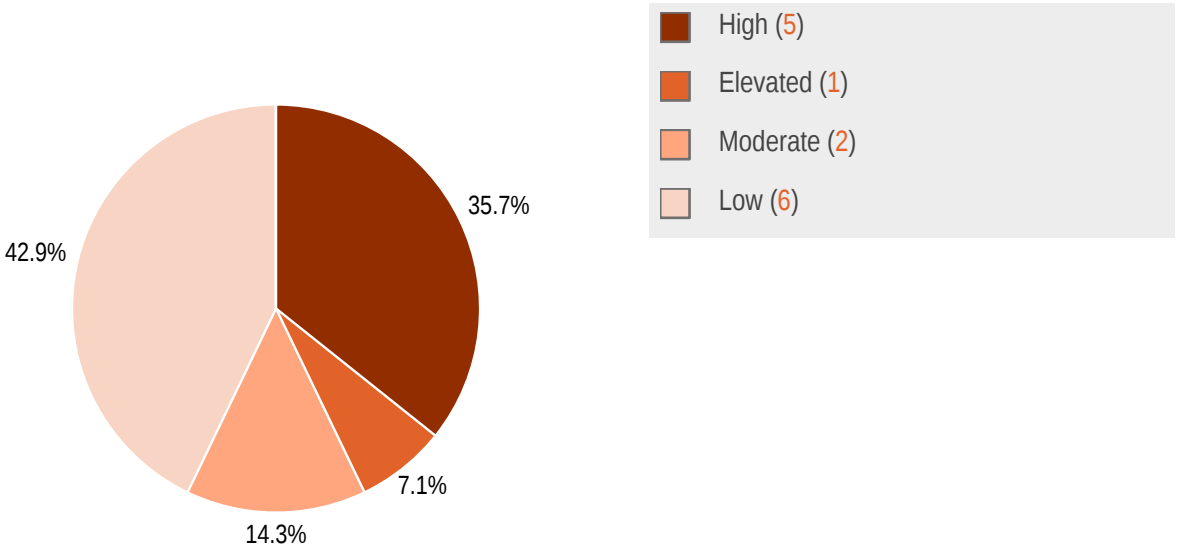
Advisory GHSA-4grw-m28r-q285 (CVE-2024-53857) covers potential resource exhaustion attacks via RPG-008 (page 25) and RPG-023 (page 21).

1.8 Summary of Findings

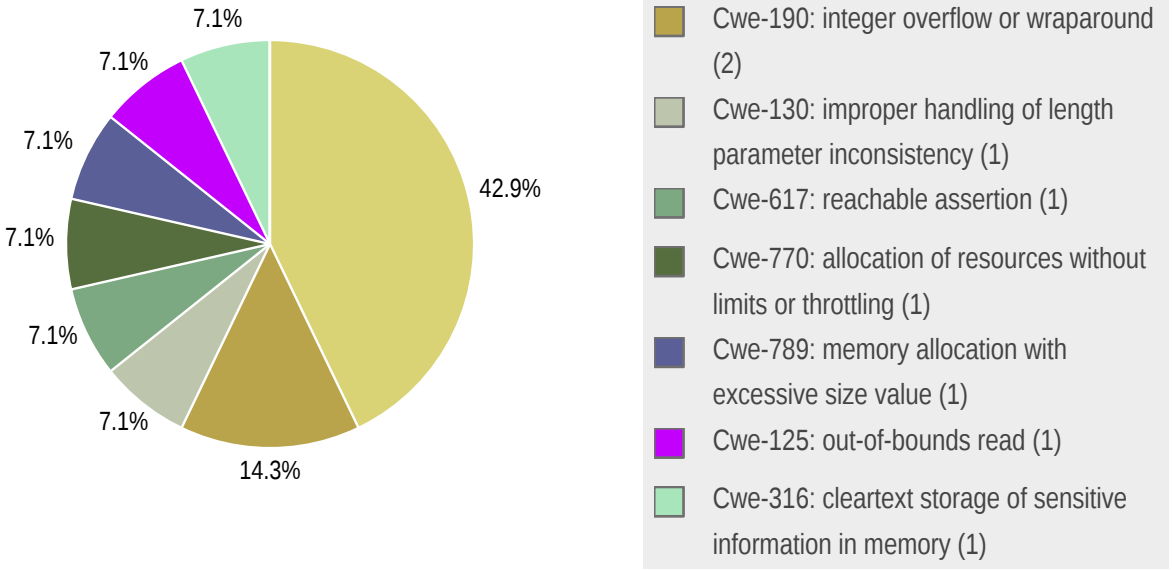
ID	Type	Description	Threat level
RPG-007	CWE-130: Improper Handling of Length Parameter Inconsistency	The central parsing code for packets of fixed size validates a length field insufficiently before accessing data, causing a program crash on malformed inputs.	High
RPG-015	CWE-248: Uncaught Exception	The handling code for signed cleartext messages crashes on some inputs.	High
RPG-016	CWE-617: Reachable Assertion	On rPGP versions before v0.12.0-alpha.3, the parsing code for packets can halt due to a violated assertion, causing a program crash on malformed inputs.	High
RPG-019	CWE-248: Uncaught Exception	The message decryption operation crashes on some malformed inputs. In one discovered variant, this happens independently of the decryption key used by the victim.	High
RPG-022	CWE-248: Uncaught Exception	The message parsing for "armor" related formats crashes on some inputs in older rPGP releases.	High
RPG-023	CWE-770: Allocation of Resources Without Limits or Throttling	The optional hashing algorithm of the string-to-key mechanism can be used to delay processing or crash the application when decrypting some message types or unlocking encrypted private keys.	Elevated
RPG-008	CWE-789: Memory Allocation with Excessive Size Value	The packet parser accepts long sequences of partial packets and allocates memory based on externally controlled length fields. During processing, this can cause rPGP to allocate multiple gigabytes of additional memory beyond the intended limit, which can lead to resource exhaustion and crashes on some systems.	Moderate
RPG-017	CWE-248: Uncaught Exception	The public key handling code can crash on some inputs.	Moderate
RPG-009	CWE-190: Integer Overflow or Wraparound	The message parsing code can trigger a problematic arithmetic edge case when parsing external inputs.	Low
RPG-010	CWE-190: Integer Overflow or Wraparound	The signature parsing code can trigger a problematic arithmetic edge case when parsing external inputs.	Low
RPG-018	CWE-248: Uncaught Exception	Several type conversion functions have a problematic design. Instead of propagating error results to the caller, they either succeed or crash the program, making them dangerous to use.	Low
RPG-020	CWE-248: Uncaught Exception	The cryptographic signing operations can crash when operating on a malformed secret key.	Low

RPG-021	CWE-125: Out-of-bounds Read	The pgp encryption operations can crash when operating with a malformed secret key.	Low
RPG-024	CWE-316: Cleartext Storage of Sensitive Information in Memory	For security reasons, sensitive data in memory should be overwritten before deallocation to reduce the risk of unintentional exposure via other vulnerabilities. rPGP already does this in some situations, but does not yet cover all relevant cases.	Low

1.8.1 Findings by Threat Level



1.8.2 Findings by Type



1.9 Summary of Recommendations

ID	Type	Recommendation
RPG-007	CWE-130: Improper Handling of Length Parameter Inconsistency	• Perform explicit length checks and reject invalid inputs. • Investigate related parsing functions for similar issues.
RPG-015	CWE-248: Uncaught Exception	• Ensure message parsers are robust against malformed inputs. • Perform more adversarial testing of parsing components.
RPG-016	CWE-617: Reachable Assertion	• Update to newer versions that no longer have this issue. • Perform more adversarial testing of parsing components.
RPG-019	CWE-248: Uncaught Exception	• Make the function more robust against malformed or malicious inputs.
RPG-022	CWE-248: Uncaught Exception	• Update to newer versions that no longer have this issue. • Perform more adversarial testing of parsing components.
RPG-023	CWE-770: Allocation of Resources Without Limits or Throttling	• Consider applying default limits on the maximum memory consumption and CPU usage. • If necessary, provide a separate API endpoint with custom limits or without limitations.
RPG-008	CWE-789: Memory Allocation with Excessive Size Value	• Abort message processing if the claimed length does not match the size of the supplied data exactly. • Consider restricting the maximum allowed buffer size, if the corresponding reduction of functionality is acceptable. • Investigate if it is possible to process large messages with a reduced memory footprint.

RPG-017	CWE-248: Uncaught Exception	• Ensure processing functions are robust against malformed inputs. • Perform more adversarial testing of conversion functions.
RPG-009	CWE-190: Integer Overflow or Wraparound	• Fix the code to avoid triggering the problematic edge cases. • Perform more testing with `overflow-checks` enabled. • Investigate other options to catch issues of this pattern during development, for example via special Rust clippy lint warnings if available.
RPG-010	CWE-190: Integer Overflow or Wraparound	• Fix the code to avoid triggering the problematic edge cases. • Perform more testing with `overflow-checks` enabled. • Investigate other options to catch issues of this pattern during development, for example via special Rust clippy lint warnings if available.
RPG-018	CWE-248: Uncaught Exception	• Consider improving the API behavior by offering less error-prone conversion functions. • Clearly document the expected crash behavior in all cases.
RPG-020	CWE-248: Uncaught Exception	• Make the signing code more robust against failure conditions.
RPG-021	CWE-125: Out-of-bounds Read	• Make the encryption code more robust against failure conditions, for example by validating and rejecting invalid keys earlier. • Investigate more systematic ways to detect code problems with integer overflow conditions.
RPG-024	CWE-316: Cleartext Storage of Sensitive Information in Memory	• Extend the use of `zeroize` to sensitive strings and functions that manipulate raw key material. • Evaluate whether `zeroize` is correctly activated in all relevant dependencies. • Evaluate whether material other than the key should also be considered sensitive. • Improve the public documentation on what `rPGP` considers sensitive.

2 Methodology

2.1 Planning

Our general approach during code audits is as follows:

During the code audit we verify if the proper security controls are present, work as intended and are implemented correctly. If vulnerabilities are found, we determine the threat level by assessing the likelihood of exploitation of this vulnerability and the impact on the Confidentiality, Integrity and Availability (CIA) of the system. We will describe how an attacker would exploit the vulnerability and suggest ways of fixing it.

This requires an extensive knowledge of the platform the application is running on, as well as the extensive knowledge of the language the application is written in and patterns that have been used. Therefore a code audit is done by specialists with a strong background in programming.

During this code audit, we take the following approach:

1. **Thorough comprehension of functionality**
   We try to get a thorough comprehension of how the application works and how it interacts with the user and other systems. Having detailed documentation at this stage is very helpful, as it aids the understanding of the application.
2. **Comprehensive code reading**
   Goals of the comprehensive code reading are:
   - to get an understanding of the whole code
   - identify adversary controlled inputs and trace their paths
   - identify issues
3. **Automated code scans**
   Using specialized tools, we scan the codebase for common issue patterns and then manually review the flagged code positions for potential negative security implications.

2.2 Risk Classification

Throughout the report, vulnerabilities or risks are labeled and categorized according to the Penetration Testing Execution Standard (PTES). For more information, see: http://www.pentest-standard.org/index.php/Reporting

These categories are:

- **Extreme**
  Extreme risk of security controls being compromised with the possibility of catastrophic financial/reputational losses occurring as a result.

- **High**

  High risk of security controls being compromised with the potential for significant financial/reputational losses occurring as a result.
- **Elevated**

  Elevated risk of security controls being compromised with the potential for material financial/reputational losses occurring as a result.
- **Moderate**

  Moderate risk of security controls being compromised with the potential for limited financial/reputational losses occurring as a result.
- **Low**

  Low risk of security controls being compromised with measurable negative impacts as a result.

3 Reconnaissance and Fingerprinting

We were able to gain information about the software and infrastructure through the following automated scans. Any relevant scan output will be referred to in the findings.

- semgrep – https://semgrep.dev
- semgrep ruleset – https://github.com/semgrep/semgrep-rules
- semgrep ruleset – https://github.com/trailofbits/semgrep-rules
- cargo audit – https://github.com/rustsec/rustsec/blob/main/cargo-audit
- cargo fuzz – https://github.com/rust-fuzz/cargo-fuzz

4 Findings

We have identified the following issues:

4.1 RPG-007 — PacketParser crashes when processing malformed message

Vulnerability ID: RPG-007	**Status:** Resolved
Vulnerability type: CWE-130: Improper Handling of Length Parameter Inconsistency	
Threat level: High	

Description:

The central parsing code for packets of fixed size validates a length field insufficiently before accessing data, causing a program crash on malformed inputs.

Technical description:

The `rPGP` packet parser trusts a length field in the input and uses it for a slice operation without verifying first that the necessary number of data bytes for the slice are actually present. For malformed inputs, this causes the Rust program to `panic` on an out-of-bounds read access, terminating the process.

Crash message:

```
thread '<unnamed>' panicked at src/packet/many.rs:120:70

range end index 52 out of range for slice of length 1
```

We confirmed that the problematic code functionality is reachable via multiple functions that process external, untrusted input:

- `Message::from_bytes()`
- `Message::from_string()`
- `Message::from_armor_single()`
- `pgp::composed::SignedPublicKey::from_bytes()`

(This list is not exhaustive)

Additional characteristics:

- Problematic inputs can be very short (2 bytes)

- The attack is independent of victim configuration (public or private keys, identities)
- The crash is reliable and happens on all platforms
- The affected parser function is in use for many formats and code paths

Fortunately, the design of the Rust language ensures that this problematic data access is reliably mitigated into a denial of service without impacts to the confidentiality or integrity of program data.

The problematic code was introduced with https://github.com/rpgp/rpgp/commit/3b71f41ebbe180af42b830b65e0b511362e11e1e and first released with `v0.12.0-alpha.3`. The first stable release affected is `v0.13.0`.

Notably, RPG-016 (page 17) and RPG-022 (page 20) are similar message parser crash findings in older `rPGP` versions.

Reproducer:

```
use pgp::composed::{Deserializable, Message};

#[test]
fn message_parser_panic1() {
    let bad_input: &[u8] = &[0xff, 0x1];

    // expected behavior
    // [...] panicked at src/packet/many.rs:120:70:
    // range end index 1 out of range for slice of length 0
    let _ = Message::from_bytes(bad_input);
}
```

Reproducer:

```
#[test]
fn message_from_armor_single_panic1() {
    let bad_input: &[u8] = &[
        45, 45, 45, 45, 45, 66, 69, 71, 73, 78, 32, 80, 71, 80, 32, 77, 69, 83, 83, 65, 71, 69, 45,
        45, 45, 45, 45, 10, 54, 84, 54, 53, 45, 45, 45, 45, 45, 69, 78, 68, 32, 80, 71, 80, 32, 77,
        69, 83, 83, 65, 71, 69, 45, 45, 45, 45, 45,
    ];

    // expected bug behavior
    // thread '<unnamed>' panicked at [..]/src/packet/many.rs:120:70:
    // range end index 62 out of range for slice of length 1
    let _ = Message::from_armor_single(bad_input);
}
```

Impact:

- Parsing of crafted inputs results in a program crash, which represents a persistent denial of service impact.

- Processing external untrusted content such as messages and packets can trigger this issue, making it easily triggered by local or remote attackers.

Recommendation:

- Perform explicit length checks and reject invalid inputs.
- Investigate related parsing functions for similar issues.

4.2 RPG-015 — Cleartext message parser crashes on malformed input

Vulnerability ID: RPG-015 **Status:** Resolved

Vulnerability type: CWE-248: Uncaught Exception

Threat level: High

Description:

The handling code for signed cleartext messages crashes on some inputs.

Technical description:

Crafted message inputs in the "ASCII armor" format can trigger a state-handling bug in the armor parsing component by omitting expected format markers.

Error message:

```
thread '<unnamed>' panicked at src/armor/reader.rs:367:13:
can only be called when done
```

Reproducers:

```
#[test]
fn cleartext_signed_message_from_armor_panic1() {
    let bad_input: &[u8] = &[
        45, 45, 45, 45, 45, 66, 69, 71, 73, 78, 32, 80, 71, 80, 32, 83, 73, 71, 78, 69, 68, 32, 77,
        69, 83, 83, 65, 71, 69, 45, 45, 45, 45, 45, 10, 10, 22, 10, 45, 45, 45, 45, 45, 66, 69, 71,
        73, 78, 32, 80, 71, 80, 32, 83, 73, 71, 78, 65, 84, 85, 82, 69, 45, 45, 45, 45, 45, 10, 72,
    ];
    let _ = pgp::composed::cleartext::CleartextSignedMessage::from_armor(bad_input);
}

#[test]
fn cleartext_signed_message_from_string_panic1() {
```

```
    // this triggers the same bug as the from_armor() case, but is more human readable

    let bad_input = "-----BEGIN PGP SIGNED MESSAGE-----\n\n-\n-----BEGIN PGP SIGNATURE-----\n-";
    let _ = pgp::composed::cleartext::CleartextSignedMessage::from_string(bad_input);
}
```

Impact:

- Parsing of crafted inputs results in a program crash, representing a persistent denial of service impact.
- The crash is reachable without supplying a valid signature.
- Signed cleartext messages are often provided by external, untrusted sources.

Recommendation:

- Ensure message parsers are robust against malformed inputs.
- Perform more adversarial testing of parsing components.

4.3 RPG-016 — PacketParser crashes due to violated assertion on malformed message

Vulnerability ID: RPG-016	**Status:** Resolved
Vulnerability type: CWE-617: Reachable Assertion	
Threat level: High	

Description:

On `rPGP` versions before `v0.12.0-alpha.3`, the parsing code for packets can halt due to a violated assertion, causing a program crash on malformed inputs.

Technical description:

When processing specially crafted inputs, a code assertion is violated and triggers a crash.

In Rust, assertions are relevant for both debug- and release-oriented build profiles, which is different from other languages. See the Rust documentation:

Assertions are always checked in both debug and release builds, and cannot be disabled.

The crash is therefore also reachable on production releases.

This finding is conceptually very similar to the denial-of-service issue RPG-007 (page 14). As far as we're aware, the code refactoring that fixed the assertion problem described here also introduced RPG-007 (page 14) at the same time. Due to the differing technical details, separate affected version ranges, and different crashing inputs, we're treating these vulnerabilities as separate findings.

Reproducer:

```
#[test]
fn message_parser_panic2() {
    // expected bug behavior:
    // thread '<unnamed>' panicked at 'assertion failed: length > 0', src/packet/many.rs:140:17:

    let bad_input: &[u8] = &[0xb7];
    let _ = Message::from_bytes(bad_input);
}
```

The problematic code was removed as part of commit 3b71f41ebbe180af42b830b65e0b511362e11e1e. The last affected crate release is `v0.12.0-alpha.2`, and the last affected stable release is `v0.11.0`.

We confirmed the reproducer crash in versions `0.11.0` and `0.8.0`. Note: We tested with Rust version 1.74.0 to avoid conflicting dependency issues.

Prior versions could not be tested due to dependency issues, namely "yanked" (removed) versions of software dependencies such as the `aes` and `aes-soft` crates and incompatibilities with newer versions.

Impact:

- Parsing of crafted inputs results in a program crash, representing a persistent denial of service impact.
- Processing external untrusted content such as messages can trigger this issue, leading us to think that this issue is reachable by remote attackers for programs that use the affected function.

Recommendation:

- Update to newer versions that no longer have this issue.
- Perform more adversarial testing of parsing components.

4.4 RPG-019 — decrypt_with_password() operation crashes on malformed messages

Vulnerability ID: RPG-019	**Status:** Resolved
Vulnerability type: CWE-248: Uncaught Exception	
Threat level: High	

Description:

The message decryption operation crashes on some malformed inputs. In one discovered variant, this happens independently of the decryption key used by the victim.

Technical description:

This finding covers two different vulnerability variants. The first variant is independent of the symmetric decryption key used by the victim, while the second variant has some dependence on the key, as shown in the code examples.

Reproducer:

```
#[test]
fn message_decrypt_with_password_panic1() {
    let bad_input: &[u8] = &[
        140, 159, 4, 1, 0, 0, 0, 167, 167, 167, 167, 167, 167, 167, 167, 0, 0, 0, 0, 0, 0, 0, 145,
        68, 32, 70, 73, 76, 69, 208, 0, 0, 0, 0, 227, 167, 167, 76, 69, 210, 69, 208, 210, 167,
        167, 167, 227, 167, 167, 76, 69, 210, 167, 167, 167, 167, 167, 167, 227, 167, 167, 76, 69,
        210, 69, 208, 210, 167, 167, 167, 227, 167, 167, 76, 69, 227, 167, 167, 76, 69, 1, 0, 0, 0,
        0, 0, 4, 184, 167, 167, 167, 227, 167, 167, 76, 69, 167, 167, 167, 167, 167, 167, 68, 32,
        70, 73, 76, 69, 208, 210, 167, 167, 167, 227, 167, 167, 76, 69, 210, 69, 208, 210, 167,
        167, 167, 227, 167, 167, 76, 69, 227, 167, 167, 69, 73, 76, 69, 208, 210, 167, 167, 167,
        227, 167, 167, 76, 69, 210, 69, 208, 210, 167, 167, 167, 227, 167, 167, 76, 69, 227, 167,
        167,
    ];
    let message = Message::from_bytes(bad_input).unwrap();

    // expected bug behavior
    // thread '<unnamed>' panicked at library/alloc/src/raw_vec.rs:545:5:
    // capacity overflow
    let _ = message.decrypt_with_password(|| "password does not matter".into());
}
```

Reproducer:

```
#[test]
fn message_decrypt_with_password_panic2() {
    let bad_input: &[u8] = &[
        0xc3, 0x20, 0x04, 0x01, 0x01, 0x02, 0x32, 0xf6, 0xe3, 0xff, 0xff, 0xac,
        0xa7, 0xa7, 0xa7, 0xff, 0xff, 0xa7, 0x26, 0xaf, 0x20, 0x4b, 0xaf, 0xa7,
        0xa7, 0xa7, 0xa7, 0xd1, 0x22, 0xa7, 0xa7, 0xa7, 0x00, 0xa7, 0xa7, 0xd1,
        0x22, 0xff, 0xff, 0xff, 0xa7, 0x26, 0xaf, 0x20, 0x4b, 0xaf
    ];
```

```
    let message = Message::from_bytes(bad_input).unwrap();

    // note that for this crash, the password does matter
    // expected bug behavior
    // thread '[..]' panicked at [..]/src/crypto/sym.rs:265:52:
    // not implemented: CFB resync is not here
    let _ = message.decrypt_with_password(|| "bogus_password".into());
}
```

Impact:

- Attempting to decrypt crafted messages can result a program crash.
- In one of the two discovered variants, the attacker does not need to know the symmetric decryption key used by the victim to trigger the issue, making them easier to attack.

Recommendation:

- Make the function more robust against malformed or malicious inputs.

4.5 RPG-022 — MessageParser crashes due to invalid state on malformed messages

Vulnerability ID: RPG-022	**Status:** Resolved
Vulnerability type: CWE-248: Uncaught Exception	
Threat level: High	

Description:

The message parsing for "armor" related formats crashes on some inputs in older `rPGP` releases.

Technical description:

During selective testing of the older `rPGP` release `0.11.0`, we observed a crash that was similar to RPG-007 (page 14) and RPG-016 (page 17). Due to the review focus on the newest `rPGP` release `0.14.0` where this issue no longer occurs, we did not investigate it extensively, but expect that it may be relevant to projects that use this older version of `rPGP`.

We observed crashes using `rPGP` versions `0.11.0` and `0.10.1`; This list is not exhaustive.

Reproducer:

```
#[test]
fn message_from_armor_single_panic2() {
    // expected bug behavior:
    // thread '[..]' panicked at [..]/src/armor/reader.rs:489:13:
    // invalid state

    let bad_input: &[u8] = b"-----BEGIN PGP SIGNATURE-----\n00LL";
    let _ = Message::from_armor_single(std::io::Cursor::new(bad_input));
}
```

Impact:

- Parsing of crafted messages results in a program crash, representing a persistent denial of service (DOS) impact.
- The affected function handles externally provided untrusted inputs.

Recommendation:

- Update to newer versions that no longer have this issue.
- Perform more adversarial testing of parsing components.

4.6 RPG-023 — Resource exhaustion via RFC9580 S2K with excessive Argon2 hash settings

Vulnerability ID: RPG-023	**Status:** Unresolved
Vulnerability type: CWE-770: Allocation of Resources Without Limits or Throttling	
Threat level: Elevated	

Description:

The optional hashing algorithm of the string-to-key mechanism can be used to delay processing or crash the application when decrypting some message types or unlocking encrypted private keys.

Technical description:

OpenPGP has a mechanism called `String-to-Key` (S2K) to derive keys from passphrases, which is based on hashing algorithms in different configurations. RFC9580 introduced a new mode based on the Argon2 key derivation

function, which is meant to provide a stronger defense against brute-force attacks by increasing the resource cost per attempted derivation.

When called to decrypt a `Symmetric Key Encrypted Session Key` (SKESK) packet which specifies the `Argon2` derivation mode, `rPGP` will attempt to reserve the required memory and perform the necessary number of iterations without applying any additional limits on these values. Unfortunately, the RFC9580 specification allows the sender to specify very large values, likely to be more future-proof in light of expected advances in computer technology over the coming decades. This creates a potential for intentional or accidental denial-of-service situations. Due to the design of Argon2, the receiver has to follow the sender's exact usage instructions for the Argon2 configuration to correctly decrypt the packet, therefore this cannot easily be mitigated without an impact on allowed functionality.

The specification describes:

> The memory size m is 2^encoded_m kibibytes (KiB) of RAM. The encoded memory size MUST be a value from 3+ceil(log2(p)) to 31, such that the decoded memory size m is a value from 8*p to 2^31. Note that memory-hardness size is indicated in KiB, not octets.

Therefore the maximum requested memory for the Argon2 computation with `encoded_m = 31` is `2^31 * 1024 Byte = 2 TiB` of memory, which exceeds the available main memory on today's typical mobile and PC environments of end users by one to three orders of magnitude.

Additionally, an attacker can set larger than usual values for the `number of passes t` and `degree of parallelism p`, which do not increase the memory requirements but slow down the overall computation process significantly.

We experimentally reproduced and confirmed relevant impacts for scenarios of both decrypting a crafted encrypted private key, and decrypting an externally provided encrypted SKESK message. Depending on the computation environment and available resources, observed effects include:

- Application crash when exceeding available memory resources
- Long computation duration of several minutes on a fast desktop computer, notably with high single CPU core usage
- Long computation consuming a large amount of memory

Since the S2K parameters are not encrypted, attackers do not need to know the symmetric key (passphrase) which is used by the victim to attempt decryption, and the decryption does not perform additional checks such as signature verification before invoking the resource-intensive Argon2 function. Attackers are therefore free to craft generic messages, or manipulate genuine SKESK messages in transit when in a position to do so.

While discussing potential mitigation strategies, the `rPGP` developers also noted that a single OpenPGP message can contain multiple encrypted packets, which likely allows triggering the affected decryption functionality sequentially multiple times, adding to the necessary computation time. We did not test this experimentally, but see it as generally plausible, complicating effective mitigation options.

The `rPGP` team introduced additional limits via security patches, which reduce the worst-case impact of individual packets. However, at the time of the review, we see the available mitigations as incomplete and therefore still mark this finding as **unresolved**.

Note that this finding is related to RPG-019 (page 19), which describes other problems in the message decryption functionality. It is also related to RPG-008 (page 25), but has a different worst-case behavior, attack conditions, and is relevant for other functions.

Reproducer:

```
#[test]
fn message_decrypt_with_password_oom_panic1() {
    let bad_input: &[u8] = &[
        195, 32, 4, 1, 4, 39, 0, 0, 0, 0, 0, 0, 0, 71, 73, 78, 32, 18, 83, 151, 2, 255, 255, 31, 22,
        22, 22, 22, 4, 172, 0, 167, 167, 167, 167,
    ];
    let message = Message::from_bytes(bad_input).unwrap();

    // expected bug behavior
    // crash
    // memory allocation of 2199023124480 bytes failed
    let _ = message.decrypt_with_password(|| "password does not matter".into());
}
```

This represents the worst-case for a remotely received packet, with all three cost factors set to maximum values as allowed by the specification.

Reproducer:

```
#[test]
fn signed_secret_key_create_signature_oom_crash2() {
    let bad_input: &[u8] = &[
        0x97, 0x04, 0x00, 0x00, 0x08, 0x29, 0x18, 0xfd, 0xff, 0x03, 0x04, 0x00, 0x00, 0x00, 0xff,
        0x7f, 0x00, 0x01, 0x4e, 0x4e, 0x4e, 0x4e, 0x11, 0x00, 0x00, 0x00, 0x00, 0x10, 0x10, 0x17,
        0x10, 0x10, 0x10, 0x10, 0x10, 0x00, 0x00, 0x00, 0x4e, 0x4e, 0x4e, 0x4e, 0xb1, 0x4e, 0xa8,
        0x4e, 0x0e, 0x4e, 0x4e, 0x4e, 0x4e, 0x4e, 0x4e, 0x4e, 0x4e, 0x4e, 0x00, 0x00,
    ];

    let dummy_data: &[u8] = &[0];

    let key = pgp::composed::SignedSecretKey::from_bytes(bad_input).unwrap();

    // expected bug behavior:
    // process allocates over 8300MB of memory and computes for dozens of seconds
    //
    // Debug print for argon2 object:
    // Argon2 { algorithm: Argon2id, version: V0x13, params:
    //  Params { m_cost: 8388608, t_cost: 16, p_cost: 16, keyid: KeyId
    //  { bytes: [0, 0, 0, 0, 0, 0, 0, 0], len: 0 }, data: AssociatedData
    //  { bytes: [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    //  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0], len: 0 }, output_len: Some(32) }, .. }

    let _ = key.create_signature(
        || "pw".into(),
        pgp::crypto::hash::HashAlgorithm::SHA2_256,
        dummy_data,
    );
```

```
}
```

Other variant:

```
// more extreme case with m_cost: 8388608, t_cost: 255, p_cost: 255
    let bad_input: &[u8] = &[
        0x97, 0x04, 0x00, 0x00, 0x08, 0x29, 0x18, 0xfd, 0xff, 0x03, 0x04, 0x00, 0x00, 0x00, 0xff,
        0x7f, 0x00, 0x01, 0x4e, 0x4e, 0x4e, 0x4e, 0x11, 0x00, 0x00, 0x00, 0x00, 0xff, 0xff, 0x17,
        0x10, 0x10, 0x10, 0x10, 0x10, 0x00, 0x00, 0x00, 0x4e, 0x4e, 0x4e, 0x4e, 0xb1, 0x4e, 0xa8,
        0x4e, 0x0e, 0x4e, 0x4e, 0x4e, 0x4e, 0x4e, 0x4e, 0x4e, 0x4e, 0x4e, 0x00, 0x00,
    ];
```

Reproducer:

```
use pgp::{types::SecretKeyTrait, Deserializable};

#[test]
fn signed_secret_key_create_signature_oom_crash4() {
    // worst case with m_cost: 2147483648, t_cost: 255, p_cost: 255
    let bad_input: &[u8] = &[
        0x97, 0x04, 0x00, 0x00, 0x08, 0x29, 0x18, 0xfd, 0xff, 0x03, 0x04, 0x00, 0x00, 0x00, 0xff,
        0x7f, 0x00, 0x01, 0x4e, 0x4e, 0x4e, 0x4e, 0x11, 0x00, 0x00, 0x00, 0x00, 0xff, 0xff, 0x1f,
        0x10, 0x10, 0x10, 0x10, 0x10, 0x00, 0x00, 0x00, 0x4e, 0x4e, 0x4e, 0x4e, 0xb1, 0x4e, 0xa8,
        0x4e, 0x0e, 0x4e, 0x4e, 0x4e, 0x4e, 0x4e, 0x4e, 0x4e, 0x4e, 0x4e, 0x00, 0x00,
    ];

    let dummy_data: &[u8] = &[0];

    let key = pgp::composed::SignedSecretKey::from_bytes(bad_input).unwrap();

    // expected bug behavior:
    // process attempts to allocate 2 TiB (!) of memory and crashes with
    // memory allocation of 2199023124480 bytes failed

    let _ = key.create_signature(
        || "pw".into(),
        pgp::crypto::hash::HashAlgorithm::SHA2_256,
        dummy_data,
    );
}
```

Impact:

- Attempting to decrypt crafted messages can result a program crash, slow processing or system instability via resource exhaustion.
- The relevant message parameters which trigger the denial of service are not encrypted or otherwise protected and therefore directly controllable by attackers without knowledge of the symmetric key used by the victim.
- Attackers can select slow processing via large iteration rounds parameter and large memory usage via the memory exponent parameter.

Recommendation:

- Consider applying default limits on the maximum memory consumption and CPU usage.
- If necessary, provide a separate API endpoint with custom limits or without limitations.

4.7 RPG-008 — PacketParser out-of-memory condition

Vulnerability ID: RPG-008	**Status:** Resolved
Vulnerability type: CWE-789: Memory Allocation with Excessive Size Value	
Threat level: Moderate	

Description:

The packet parser accepts long sequences of partial packets and allocates memory based on externally controlled length fields. During processing, this can cause `rPGP` to allocate multiple gigabytes of additional memory beyond the intended limit, which can lead to resource exhaustion and crashes on some systems.

Technical description:

We found this issue during fuzz testing of `Message::from_bytes(data)`.

The OpenGPG message standard allows multiple packet formats, which have different upper limits on their maximum size. In several scenarios, attackers can use this to force or trick `rPGP` in allocating large amounts of memory during message parsing, which may cause a crash or other system instability.

For the purposes of this attack, there are two relevant formats:

- Packets of fixed size can specify a message body length of up to `2^32 Byte`, which is 4GiB. (RFC9580 `5-octet Body Length header`)
- Partial packets can consist of an arbitrary number of segments with up to `2^30 Byte` each, which is 1GiB. (RFC9580 `Partial Body Length headers`)

There are additional technical details and requirements, omitted here for brevity.

The attack has two variable components.

First, the attacker can chain an arbitrary number of the partial packet segments. Note that the specified length of data in this packet type has to actually be present for the victim to read in order for the attack to succeed, meaning that the transmitted attack message grows by this amount.

Second, the attacker can finish the stream of partial messages with one fixed message with a large claimed size. The analyzed version of `rPGP` accepts the specified message length parsed via `read_packet_len()` and then reserves a message buffer of this size in memory, without checking if this amount of data was actually provided by the attacker. The specified length of data in this packet type therefore does not increase the attack message size.

Generally speaking, we foresee two main scenarios:

1. The attacker is limited in the length of the overall attack message, for example due to external restrictions on the length of an email or other communication channel. They chain a single short partial packet (with actual data) with a fixed packet (with claimed but not provided data). Using this combination, they trigger ca. 4GiB of memory allocation with only 519 Byte of transmitted total message size.
2. The attacker is not limited in the length of the overall attack message, for example when providing a file in a file system, or otherwise unbounded transfers. They chain an arbitrary number of N long partial packets (up to 1GiB each) and end the message with a fixed packet (with claimed but not provided data). Using this combination, they trigger up to `(N * 1 GiB) + 4 GiB` of memory allocation with about `N * 1 GiB` of transmitted total message size.

We were able to build proof-of-concept variants for both attacks.

There are several considerations for defenses and partial mitigations of this attack. The primary and easiest improvement we see as possible for `rPGP` is to validate the claimed length of fixed messages, or only allocate buffer memory as it is actually needed, in order to remove the attacker's capability to trigger large allocations with small messages.

The secondary and less straightforward mitigation is to apply some form of additional limit on the maximum allocated memory in each case to reduce the risk of denial-of-service attacks. Note that this comes at the cost of compatibility with legitimate use cases that involve large but valid OpenGPG messages. Since `rPGP` already applies a hard limit of `1GiB` for another related program flow that involves messages without a fixed size, see https://github.com/rpgp/rpgp/issues/158, applying similar limits for all message handling paths would reduce the worst-case impact and serve as a partial mitigation. Since this also restrict the capabilities of the library, we recommend considering viable limits and potential ways for developers to opt-out of the default limits.

During the review phase, the `rPGP` developers introduced additional length handling which limits all messages to a total input size of 1 GiB. While additional handling improvements may be possible, we see this as a sufficient security patch to mitigate the main security impact.

Technical notes:

Internally, `rPGP` uses the `usize` variable type for the length field which can grow to larger values on most modern architectures, but both the `old_packet_header()` and `new_packet_header()` code variants do not assign length values greater than `2^32` for any individual packet, which is due to corresponding limits in the OpenPGP specification.

Code snippet of assignment via `u32` Rust integer type:

```
// Five-Octet Lengths
        255 => {
```

```
            let (i, len) = be_u32(i)?;
            Ok((i, (len as usize).into()))
        }
```

Stacktrace snippet captured during parsing of problematic input:

```
[...]
#4  pgp::packet::many::read_fixed<&[u8]> (reader=<optimized out>, len=4294967294,
 out=0x7fffffffcae0) at src/packet/many.rs:247
[...]
```

Relevant code of buffer resizing:

```
fn read_fixed<R: Read>(
    reader: &mut BufReader<R, MinBuffered>,
    len: usize,
    out: &mut Vec<u8>,
) -> Result<()> {
    let out_len = out.len();
    out.resize(out_len + len, 0u8);
    reader.read_exact(&mut out[out_len..])?;

    Ok(())
}
```

Reproducer:

```
#[test]
fn message_from_bytes_oom1() {
    let bad_input: &[u8] = &[
        0xc8, 0xe9, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
        0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
        0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
        0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
        0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
        0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
        0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
        0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
        0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
        0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
        0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
        0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
        0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
        0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
        0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
        0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
        0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
        0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
        0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
        0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
        0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
        0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
        0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
        0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
        0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
        0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
        0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
        0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
        0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
```

```
        0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
        0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
        0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
        0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
        0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
        0x00, 0x00, 0x00, 0x00, 0xff, 0xff, 0xff, 0xff, 0xff,
    ];

    // expected bug behavior
    // input describes a partial message (512 Byte) + a fixed size message (close to 2^32 Byte)
    //
    // uses over 4GiB of memory, which may crash on some systems
    // slow processing (several seconds), depending on hardware details and Rust optimization
    let _ = Message::from_bytes(bad_input).unwrap();
}
```

Impact:

- On resource-constrained systems or environments, this may lead to a program crash or other instabilities.
- See the technical description section for details on the different impacts and scenarios.

Recommendation:

- Abort message processing if the claimed length does not match the size of the supplied data exactly.
- Consider restricting the maximum allowed buffer size, if the corresponding reduction of functionality is acceptable.
- Investigate if it is possible to process large messages with a reduced memory footprint.

4.8 RPG-017 — SignedPublicKey as_unsigned() crashes on malformed inputs

Vulnerability ID: RPG-017 **Status:** Resolved

Vulnerability type: CWE-248: Uncaught Exception

Threat level: Moderate

Description:

The public key handling code can crash on some inputs.

Technical description:

Notably, the underlying bug was reported and analyzed publicly via https://github.com/rpgp/rpgp/issues/259 in December 2023, but was reported only for workflows that involve special forms of private keys, which have different overall security considerations.

The new report shows that this crash outcome is also reachable when exclusively handling public keys, which is a more exposed flow since public keys are more often fetched from external untrusted sources and relevant processing should be very robust.

Reproducer:

```
#[test]
fn signed_public_key_as_unsigned_panic1() {
    let bad_input: &[u8] = &[155, 4, 228, 4, 0, 4, 0];

    let key = pgp::composed::SignedPublicKey::from_bytes(bad_input).unwrap();

    // expected bug behavior:
    // thread '<unnamed>' panicked at src/composed/signed_key/shared.rs:116:35:
    // missing user ids
    let _ = key.as_unsigned();
}
```

Impact:

- Processing problematic public keys with the affected function results in a program crash, representing a persistent denial of service impact.
- It is unclear if the affected function is commonly used on untrusted input.

Recommendation:

- Ensure processing functions are robust against malformed inputs.
- Perform more adversarial testing of conversion functions.

4.9 RPG-009 — "Attempt to subtract with overflow" in message parsing

Vulnerability ID: RPG-009 **Status:** Resolved

Vulnerability type: CWE-190: Integer Overflow or Wraparound

Threat level: Low

Description:

The message parsing code can trigger a problematic arithmetic edge case when parsing external inputs.

Technical description:

We discovered the issues during fuzz testing of `Message::from_bytes(data)`.

The panics occur when running with Rust overflow checks enabled, which is the case in the `--debug` profile but not in the `--release` profile.

This finding covers at least three code positions:

- `src/packet/public_key_parser.rs:250:47`
- `src/packet/signature/de.rs:391:25`
- `src/types/params/secret.rs:106:47`

Note that this finding may partially overlap with RPG-010 (page 31), as they share some affected code paths.

Crash message:

```
thread '<unnamed>' panicked at src/packet/public_key_parser.rs:250:47:
attempt to subtract with overflow
```

Crash message:

```
thread '<unnamed>' panicked at src/packet/signature/de.rs:391:25:
attempt to subtract with overflow
```

Reproducer:

```
#[test]
fn message_from_bytes_subtract_with_overflow1() {
    let bad_input: &[u8] = &[187, 6, 227, 0, 255, 255, 255, 255, 255, 255, 255];

    // depends on "--debug" profile
    // expected bug behavior
    // thread '<unnamed>' panicked at src/packet/public_key_parser.rs:250:47:
    // attempt to subtract with overflow
    let _ = Message::from_bytes(bad_input);
}
```

Reproducer:

```
#[test]
fn message_from_bytes_subtract_with_overflow2() {
    let bad_input: &[u8] = &[139, 4, 16, 0, 0, 0, 2, 0, 0];

    // depends on "--debug" profile
    // expected bug behavior
    // thread '<unnamed>' panicked at src/packet/signature/de.rs:391:25:
    // attempt to subtract with overflow
```

```
    let _ = Message::from_bytes(bad_input);
}
```

Reproducer:

```
#[test]
fn message_from_bytes_subtract_with_overflow3() {
    let bad_input: &[u8] = &[151, 6, 7, 8, 0, 0, 0, 0, 0, 0, 0, 113, 113];

    // depends on "--debug" profile
    // expected bug behavior
    // thread '<unnamed>' panicked at src/types/params/secret.rs:106:47:
    // attempt to subtract with overflow
    let _ = Message::from_bytes(bad_input);
}
```

Impact:

- The exact impact on production builds is unclear.
- For debug builds, malformed inputs will cause a crash.
- Reachable arithmetic edge cases in cryptographic code are generally risky, since they may circumvent existing checks or assumptions.

Recommendation:

- Fix the code to avoid triggering the problematic edge cases.
- Perform more testing with `overflow-checks` enabled.
- Investigate other options to catch issues of this pattern during development, for example via special Rust clippy lint warnings if available.

4.10 RPG-010 — "Attempt to subtract with overflow" when parsing signatures

Vulnerability ID: RPG-010 **Status:** Resolved

Vulnerability type: CWE-190: Integer Overflow or Wraparound

Threat level: Low

Description:

The signature parsing code can trigger a problematic arithmetic edge case when parsing external inputs.

Technical description:

We discovered the issues during fuzz testing of `pgp::composed::StandaloneSignature::from_bytes(data)`.

The panics only occur when running with Rust overflow checks enabled, which is the case in the `--debug` profile but not in the `--release` profile.

This finding covers at least three code positions:

- `src/types/params/secret.rs:106:43`
- `src/packet/user_attribute.rs:148:45`
- `src/packet/user_attribute.rs:165:41`

Note that this finding may partially overlap with RPG-009 (page 29), as they share some affected code paths.

Crash message:

```
thread '<unnamed>' panicked at src/types/params/secret.rs:106:43:
attempt to subtract with overflow
```

Reproducer:

```
#[test]
fn standalone_signature_subtract_with_overflow1() {
    let bad_input: &[u8] = &[209, 3, 0, 252, 45];

    // expected bug behavior
    // thread '<unnamed>' panicked at src/packet/user_attribute.rs:165:41:
    // attempt to subtract with overflow
    let _ = pgp::composed::StandaloneSignature::from_bytes(bad_input);
}
```

Impact:

- The exact impact on production builds is unclear.
- For debug builds, malformed inputs will cause a crash.
- Reachable arithmetic edge cases in cryptographic code are generally risky, since they may circumvent existing checks or assumptions.

Recommendation:

- Fix the code to avoid triggering the problematic edge cases.
- Perform more testing with `overflow-checks` enabled.
- Investigate other options to catch issues of this pattern during development, for example via special Rust clippy lint warnings if available.

4.11 RPG-018 — Fragile into_*() functions can crash on some inputs

Vulnerability ID: RPG-018 **Status:** Unresolved

Vulnerability type: CWE-248: Uncaught Exception

Threat level: Low

Description:

Several type conversion functions have a problematic design. Instead of propagating error results to the caller, they either succeed or crash the program, making them dangerous to use.

Technical description:

`rPGP` offers several conversion functions that convert or extract objects of different types, and intentionally crash the program ("panic") if they cannot succeed in the conversion. In at least one case, this side effect is undocumented, which may contribute to accidental security-related bugs in programs that use the `rPGP` library.

Identified functions:

- `Message::into_signature()`
- `PublicOrSecret::into_secret()`
- `PublicOrSecret::into_public()`

Problematic code:

```
/// Convert the message to a standalone signature according to the cleartext framework.
    pub fn into_signature(self) -> StandaloneSignature {
        match self {
            Message::Signed { signature, .. } => StandaloneSignature::new(signature),
            _ => panic!("only signed messages can be converted to standalone signature messages"),
        }
    }
```

Note that the function definition does not allow returning an error. It does not have a `Result<>` result type that allows for `Ok()` and `Err()` conditional variants.

Reproducer:

```
#[test]
fn message_into_signature_panic1() {
    let bad_input: &[u8] = &[163, 0, 163];
    let message = Message::from_bytes(bad_input).unwrap();

    // expected bug behavior
    // thread '<unnamed>' panicked at src/composed/message/types.rs:587:18:
    // only signed messages can be converted to standalone signature messages
    let _ = message.into_signature();
}
```

For the two `PublicOrSecret` conversion functions, the code documentation string on the API is already clear on the potential crash outcome:

```
/// Panics if not a secret key.
    pub fn into_secret(self) -> SignedSecretKey {
        match self {
            PublicOrSecret::Public(_) => panic!("Can not convert a public into a secret key"),
            PublicOrSecret::Secret(k) => k,
        }
    }

    /// Panics if not a public key.
    pub fn into_public(self) -> SignedPublicKey {
        match self {
            PublicOrSecret::Secret(_) => panic!("Can not convert a secret into a public key"),
            PublicOrSecret::Public(k) => k,
        }
    }
```

See https://github.com/rpgp/rpgp/issues/439 and https://github.com/rpgp/rpgp/pull/436 for work on these issues. Since the pull requests were not merged at the end of the review period, we're still marking this finding as **unresolved**.

Impact:

- This represents a denial of service impact, depending on how the library is used.
- For example, running the `Message::into_signature()` function on valid but unsigned messages will result in a crash.

Recommendation:

- Consider improving the API behavior by offering less error-prone conversion functions.

- Clearly document the expected crash behavior in all cases.

4.12 RPG-020 — SignedSecretKey create_signature() crashes on malformed keys

Vulnerability ID: RPG-020 **Status:** Resolved

Vulnerability type: CWE-248: Uncaught Exception

Threat level: Low

Description:

The cryptographic signing operations can crash when operating on a malformed secret key.

Technical description:

This finding describes three separate but related code issues.

In the first variant, using a malformed private key with `rPGP` triggers a crash in the `num-bigint-dig` library via an illegal operation:

```
thread '[..]' panicked at [..]/.cargo/registry/src/index.crates.io-6f17d22bba15001f/num-bigint-dig-0.8.4/src/algorithms/sub.rs:75:5:
Cannot subtract b from a because b is larger than a.
```

See RPG-021 (page 37) for details on the patch in the upstream cryptography library `rsa` which resolves this problem.

In the second variant, using a malformed private key triggers integer overflow and subsequent assertion violations in `rPGP` while handling encrypted secrets.

In the third variant, `rPGP` attempts to allocate an excessive amount of memory, causing the program to be aborted by the operating system.

Reproducer:

```
use pgp::types::SecretKeyTrait;
#[test]
fn signed_secret_key_create_signature_panic1() {
    let bad_input: &[u8] = &[
        151, 3, 255, 251, 255, 63, 39, 254, 3, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 4,
    ];

    let dummy_data: &[u8] = &[0];

    let key = pgp::composed::SignedSecretKey::from_bytes(bad_input).unwrap();
```

```
    // expected bug behavior:
    // thread '<unnamed>' panicked at [..]/num-bigint-dig-0.8.4/src/algorithms/sub.rs:75:5:
    // Cannot subtract b from a because b is larger than a.
    let _ = key.create_signature(
        || "pw".into(),
        pgp::crypto::hash::HashAlgorithm::SHA2_256,
        dummy_data,
    );
}
```

Reproducer:

```
#[test]
fn signed_secret_key_create_signature_panic2() {
    let bad_input: &[u8] = &[
        0x97, 0x04, 0x00, 0x00, 0x08, 0x29, 0xc1, 0xfd, 0xff, 0x03, 0x03, 0x02, 0x08, 0x00, 0xf8,
        0xff, 0x00, 0xff, 0x00, 0x00, 0x00, 0x00, 0x00, 0xff, 0x00, 0xf8, 0xf8, 0xff, 0x00, 0xff,
        0x00, 0xff, 0x00,
    ];

    let dummy_data: &[u8] = &[0];

    let key = pgp::composed::SignedSecretKey::from_bytes(bad_input).unwrap();

    // expected bug behavior for --debug:
    // thread [..] panicked at [..]/src/types/params/encrypted_secret.rs:155:48:
    // attempt to subtract with overflow
    //
    // expected bug behavior for --release:
    // thread '[..]' panicked at [..]/src/types/params/encrypted_secret.rs:155:39:
    // assertion failed: mid <= self.len()

    let _ = key.create_signature(
        || "pw".into(),
        pgp::crypto::hash::HashAlgorithm::SHA2_256,
        dummy_data,
    );
}
```

Reproducer:

```
#[test]
fn signed_secret_key_create_signature_oom_crash1() {
    let bad_input: &[u8] = &[
        0x97, 0x04, 0x00, 0x00, 0x08, 0x29, 0xc1, 0xfd, 0xff, 0x9f, 0x04, 0x8f,
        0xe4, 0xff, 0xff, 0xff, 0xff, 0x80, 0x8f, 0x8f, 0x8f, 0x00, 0x01, 0x00,
        0x00, 0x00, 0xaf, 0xf8, 0x1b, 0x1b
    ];

    let dummy_data: &[u8] = &[0];

    let key = pgp::composed::SignedSecretKey::from_bytes(bad_input).unwrap();

    // expected bug behavior:
    // memory allocation of 137438871552 bytes failed

    let _ = key.create_signature(
        || "pw".into(),
```

```
        pgp::crypto::hash::HashAlgorithm::SHA2_256,
        dummy_data,
    );
}
```

Impact:

- Signing any data with a malformed private key leads to a program crash.
- Since private keys are typically chosen by the victim, this requires special circumstances to be useful for attacks.

Recommendation:

- Make the signing code more robust against failure conditions.

4.13 RPG-021 — SignedSecretKey encrypt() crash on malformed key

Vulnerability ID: RPG-021 **Status:** Resolved

Vulnerability type: CWE-125: Out-of-bounds Read

Threat level: Low

Description:

The `pgp` encryption operations can crash when operating with a malformed secret key.

Technical description:

This finding describes two separate but related code issues.

In the first variant, using a malformed private key with `rPGP` triggers an integer overflow and crash after a detected and mitigated out-of-bounds access in the `rsa` Rust library maintained by RustCrypto. The `rPGP` developer Friedel Ziegelmayer is also a developer for the `rsa` crate, and he handled the relevant analysis and submitted a pull request for a patch which improves key validation and rejection of problematic keys. This was included in `rsa` release `0.9.7`, fixing this issue variant.

The second variant is a problem in `rPGP` with insufficient error handling in the code for `X448`-type cryptographic keys.

Reproducer:

```
#[test]
fn signed_secret_key_encrypt_panic1() {
```

```
    let bad_input: &[u8] = &[
        197, 159, 4, 159, 1, 0, 20, 2, 0, 61, 0, 0, 0, 64, 0, 201, 0, 197, 0, 1, 251, 213, 0, 201,
        0, 250, 196, 0, 197, 0, 197, 0, 197, 0, 201, 0, 197, 0, 197, 0, 201, 255, 255, 255, 255,
        255, 255, 255, 5, 205, 205, 205, 205, 43, 129, 0, 0, 0, 0, 0, 0, 0, 2, 0, 0, 0, 0, 1, 1, 1,
        161, 4, 0, 242, 143, 4, 4, 135, 6, 0, 0, 0, 0, 6, 0, 0, 0, 0, 242, 143, 4, 4, 0, 0, 0, 0,
        0, 0, 0, 2, 0, 0, 0, 1, 1, 1, 161, 4, 0, 143, 4, 4, 135, 6, 0, 0, 0, 0, 4, 0, 242, 143, 4,
        4, 135, 6, 0, 0, 0, 0, 0, 0, 0, 0, 242, 143, 4, 4, 0, 0, 0, 0, 0, 0, 0, 2, 0, 0, 0, 0, 1,
        1, 143, 4, 4, 135, 6, 0, 0, 0, 0,
    ];

    // no particular meaning of this data
    let dummy_plaintext = vec![0u8; 128];

    // note, this is non-deterministic, but does not matter for reproduction
    let mut rng = rand::thread_rng();

    let key = pgp::composed::SignedSecretKey::from_bytes(bad_input).unwrap();

    // expected bug behavior on --release:
    // thread '<unnamed>' panicked at [..]/rsa-0.9.6/src/algorithms/pkcs1v15.rs:51:39:
    // range end index 18446744073709551492 out of range for slice of length 5
    //
    // expected bug behavior on --debug:
    // thread 'signed_secret_key_encrypt_panic1' panicked at [..]/rsa-0.9.6/src/algorithms/
pkcs1v15.rs:44:20:
    // attempt to subtract with overflow
    //
    // crash also happens with pgp::types::EskType::V3_4
    let _ciphertext = {
        key.encrypt(&mut rng, dummy_plaintext.as_slice(), pgp::types::EskType::V6)
    };
}
```

Reproducer:

```
#[test]
fn signed_secret_key_encrypt_panic2() {
    let bad_input: &[u8] = &[
        0x97, 0x04, 0x00, 0x1a, 0x1a, 0x1a, 0x1a, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
        0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
        0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
        0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
        0x00, 0x00, 0x00, 0x29,
    ];

    // no particular meaning of this data
    let dummy_plaintext = vec![0u8; 1];

    // note, this is non-deterministic, but does not matter for reproduction
    let mut rng = rand::thread_rng();

    let key = pgp::composed::SignedSecretKey::from_bytes(bad_input).unwrap();

    // expected bug behavior:
    // thread '[..]' panicked at [..]/src/crypto/x448.rs:149:75:
    // 56
    //
    // crash also happens with pgp::types::EskType::V3_4
    let _ciphertext = {
        key.encrypt(
```

```
            &mut rng,
            dummy_plaintext.as_slice(),
            pgp::types::EskType::V6,
        )
    };
}
```

Impact:

- Encrypting any data with a malformed private key leads to a program crash.
- Since private keys are typically chosen by the victim, this requires special circumstances for practical attacks.

Recommendation:

- Make the encryption code more robust against failure conditions, for example by validating and rejecting invalid keys earlier.
- Investigate more systematic ways to detect code problems with integer overflow conditions.

4.14 RPG-024 — Improve sanitization of sensitive data in memory after use

Vulnerability ID: RPG-024 **Status:** Unresolved

Vulnerability type: CWE-316: Cleartext Storage of Sensitive Information in Memory

Threat level: Low

Description:

For security reasons, sensitive data in memory should be overwritten before deallocation to reduce the risk of unintentional exposure via other vulnerabilities. `rPGP` already does this in some situations, but does not yet cover all relevant cases.

Technical description:

`rPGP` uses the `zeroize` library for the purpose of automatically overwriting sensitive regions in memory. The `zeroize` mechanism requires sensitive objects such as structs with private key data to be annotated with relevant deletion properties (specifically, Rust traits like `#[derive(Zeroize, ZeroizeOnDrop)]`).

The analyzed `rPGP` version has annotated a number of structs and enum objects which directly hold private key material.

However, we have not seen corresponding `zeroize` annotations on:

- Passphrase strings
- Secret keys derived by the String-to-Key (S2K) mechanism

Additionally, `rPGP` depends on several cryptographic libraries, but does not correctly activate their optional `zeroize` functionality, increasing the risk that sensitive data will stay in memory after processing with functions from those libraries.

An example of the sensitive use of the mentioned hash algorithm related libraries is the S2K functionality, which hashes sensitive passphrases to derive secret keys.

The following libraries have this hardening functionality in a stable release:

- `argon2`
  - Note that the `zeroize` feature is usable but not explicitly specified. We improved this via an upstream PR, which was accepted.
- `sha1-checked`

The following libraries have this hardening functionality, but not yet in a stable release:

- `sha1`
- `sha2`
- other hashing libraries

During the review, the `rPGP` team merged https://github.com/rpgp/rpgp/pull/440 to activate improved memory handling in dependencies. We're marking this finding as **unresolved** since more improvements are necessary to cover the described issues.

Impact:

- Other memory safety vulnerabilities in related components may, under some circumstances, leak sensitive information which otherwise would be deleted.
- This is an incomplete security hardening finding, and not a directly exploitable vulnerability.

Recommendation:

- Extend the use of `zeroize` to sensitive strings and functions that manipulate raw key material.
- Evaluate whether `zeroize` is correctly activated in all relevant dependencies.
- Evaluate whether material other than the key should also be considered sensitive.
- Improve the public documentation on what `rPGP` considers sensitive.

5 Non-Findings

In this section we list some of the things that were tried but turned out to be dead ends, or have an informational nature.

5.1 NF-001 — Improve SECURITY.md

The `rPGP` SECURITY.md file is already in good shape, and outlines relevant project details and expectations to any potential issue submitter.

However, it currently does not allow people without a GitHub account to contact the team. Even if the project has a strong preference for submissions through GitHub-specific mechanisms, we recommend specifying an email address that can be used for security submissions. If spam messages on this address become a problem, we suggest asking submitters to put a specific keyword into the email title, such as `[Vulnerability]`, which will help distinguish them from automated and unwanted messages. Given the project scope, we also recommend considering publishing a corresponding PGP public key for encrypted submissions.

Additionally, the `rPGP` project does not appear to have a public policy on which release branches or platforms receive security support. It could be helpful for dependent projects and end users to know about any maintenance goals the `rPGP` maintainers may have.

5.2 NF-002 — Investigated SHA-1 usage

The `Semgrep` static analysis tool issued the following warning:

```
src/crypto/sym.rs
    ## semgrep-rules-official.rust.lang.security.insecure-hashes
          Detected cryptographically insecure hashing function

          176# let mut digest = Sha1::new();
```

The SHA-1 message-digest algorithm suffers from cryptographic weaknesses.

The `decrypt_protected()` function decrypts and checks a special standardized OpenGPG format. According to `RFC4880` Section `5.13` and `5.14`, the use of SHA-1 is mandatory:

> The Modification Detection Code packet contains a SHA-1 hash of plaintext data, which is used to detect message modification. [..]

As such, `rPGP` needs to continue to use SHA-1 in its codebase to process these messages. We therefore consider the use of the deprecated algorithm as necessary for backwards compatibility, but recommend documenting the potential risks of accepting or falling back on deprecated cryptographic algorithms.

Notably, the `rPGP` authors are aware of the attacks on SHA-1 and have contributed a special hardened SHA-1 implementation to the RustCrypto upstream project that acts as a mitigation for the known chosen-prefix attack. They

make use of the corresponding `sha1-checked` Rust library crate within `rPGP` where relevant to mitigate some of the impact of SHA-1 weaknesses.

5.3 NF-003 — Investigated MD5 usage

The `Semgrep` static analysis tool issued the following warning:

```
src/packet/key/public.rs
    ## semgrep-rules-official.rust.lang.security.insecure-hashes
          Detected cryptographically insecure hashing function

          306# let mut h = Md5::new();
```

The MD5 message-digest algorithm suffers from extensive vulnerabilities.

The `fingerprint()` function aims to support the older `V2` and `V3` key versions, which use MD5 as per `RFC4880` section `12.2`:

> The fingerprint of a V3 key is formed by hashing the body (but not the two-octet length) of the MPIs that form the key material (public modulus n, followed by exponent e) with MD5. Note that both V3 keys and MD5 are deprecated. [..]

We therefore consider the use of the deprecated algorithm as necessary for backwards compatibility, but recommend documenting the potential risks of accepting or falling back on deprecated cryptographic algorithms.

5.4 NF-005 — Cargo audit results

```
cargo audit
    Fetching advisory database from `https://github.com/RustSec/advisory-db.git`
      Loaded 664 security advisories (from [..]/.cargo/advisory-db)
    Updating crates.io index
    Scanning Cargo.lock for vulnerabilities (243 crate dependencies)
Crate:     rsa
Version:   0.9.6
Title:     Marvin Attack: potential key recovery through timing sidechannels
Date:      2023-11-22
ID:        RUSTSEC-2023-0071
URL:       https://rustsec.org/advisories/RUSTSEC-2023-0071
Severity:  5.9 (medium)
Solution:  No fixed upgrade is available!
Dependency tree:
rsa 0.9.6
├── pgp 0.14.0
└── pgp 0.10.2
    └── pgp 0.14.0

error: 1 vulnerability found!
```

The `RUSTSEC-2023-0071` issue is relevant, but well-known to the `rPGP` authors and explicitly documented in the project. As such, it is not a new finding.

5.5 NF-006 — Evaluated relevance of RUSTSEC-2024-0344

`rPGP` uses the `curve25519-dalek` crate to perform cryptographic operations on elliptic curves.

The RUSTSEC-2024-0344 advisory covers a cryptographic side channel in `curve25519-dalek` which was fixed in July 2024 and released as the patched `4.1.3` version.

For the primarily analyzed version of `rPGP`, this dependency issue was already resolved. Prior to the review, the dependency pinning of `rPGP` already ensured that only patched versions of `curve25519-dalek` are used; see this commit.

We therefore see this as a non-finding, but were interested in the behavior for end users of older stable versions, since other consumers of the `rPGP` library may depend on these old versions.

Overview:

- `rPGP` versions `v0.10.2` and older reference `curve25519-dalek` versions that don't have a backported direct security update available, so they pull in a vulnerable `curve25519-dalek` version.
- For `rPGP` versions `v0.11.0` to `v0.13.2` (inclusive), the `cargo` update manager can switch to a patched version of `curve25519-dalek` on newer builds, but is not forced to do so.
- `rPGP` version `v0.14.0` and newer enforce the use of the patched dependency.

Due to limited evaluation time, we did not further investigate the practical relevance of the cryptographic side channel for the mentioned `rPGP` releases.

5.6 NF-011 — Improve security documentation on "hazmat" aspects of rPGP

Cryptographic libraries are often faced with the challenge that exposing low-level APIs, which need to be used in special ways to result in meaningful and secure operations, can result in harmful effects for end users in case of accidental misuse. A mismatch of the implicit security expectations or presence of complex error conditions can contribute to this problem.

Some libraries split the exposed APIs into one class that is considered safer & easy to use, and a second class that is marked as dangerous. In some cases, the latter is labeled an "hazardous materials" ("hazmat") area, for example.

After discussing this topic with the core `rPGP` developers, our impression is that the `rPGP` project intentionally aims to be a low-level library that includes such "potentially dangerous" functions which are not always safe to use for all inputs and parameters. This is also reflected in project goals for backwards compatibility with older, weaker, or

unusual cryptography standards & use cases over hardened but restrictive defaults and workflows. For example, see `IMPL_STATUS.md` in the `rPGP` project documentation for details of supported algorithms.

Publicly reported examples of this include https://github.com/rpgp/rpgp/issues/184 "Non-Encryption Subkeys May Be Used to Encrypt Data", where the flexibility on (sub-)key usage by `rPGP` may violate expectations and cause interoperability issues with other programs.

Other related topics include the handling of expiry, validity, and revocation status properties of PGP related objects.

At the time of the audit, the `rPGP` developers indicated that they're considering building a more high-level focused library layer to complement the "hazmat" nature of `rPGP` itself, but this was out of scope for the review.

We recommend extensively documenting the security expectations, properties and limitations of `rPGP` in a way that is clear to other projects and developers.

5.7 NF-013 — Unexpected message serialization roundtrip behavior

Some correctly parsed messages cannot be serialized. This is unexpected and should be documented more clearly.

In the `rPGP` logic, some messages successfully import (deserialize) to a message object, but then fail with the corresponding export (serialize) function call.

The error behavior is benign and clearly allowed by the return type, so we do not consider it a security issue. According to the developers, this behavior is well-known, as some states can be held in memory but not exported. However, downstream consumers of the library may not expect this behavior, and it is not clearly documented, which may lead to incorrect reasoning about the behavior, contributing to bugs.

We recommend improving the documentation on the serialization and de-serialization behavior.

Reproducer:

```
#[test]
fn message_valid_serialization_error_to_armor1() {
    let bad_input: &[u8] = &[132, 4, 5, 0, 163, 163, 132, 4, 5, 0, 163, 163, 132, 5, 20, 84, 0, 163,
 163, 132, 135, 3, 209, 163, 86, 209, 209, 209, 209, 209, 209, 209, 0, 0, 0, 167, 167, 167, 167,
 167, 167, 167, 167, 167, 167, 167, 167, 167, 209, 209, 167, 167, 167, 167, 167, 167, 7, 0, 0, 0,
 0, 167, 167, 209, 209, 0, 0, 0, 82, 73, 163, 86, 167, 167, 167, 167, 167, 167, 167, 167, 167, 167,
 167, 167, 209, 209, 0, 0, 0, 82, 73, 163, 86, 86, 167, 167, 167, 167, 167, 167, 167, 167, 167, 167,
 167, 167, 209, 209, 0, 0, 0, 82, 73, 163, 86, 209, 0, 0, 0, 82, 73, 163, 86, 167, 167, 167, 167,
 167, 167, 167, 167, 167, 167, 167, 167, 0, 0, 0, 0, 0, 0, 2, 32, 86, 86, 167, 167, 167, 167, 167,
 167, 167, 167, 167, 167, 167, 167, 209, 209, 0, 0, 0, 82, 73, 163, 86];

    // deserialization succeeds
    let message_deserialized = Message::from_bytes(bad_input).unwrap();

    // serialization errors out
    // `Err` value: Message("failed to write EskBytes for Unknown(209)")
    let _ = message_deserialized.to_armored_bytes(None.into()).unwrap();
}
```

5.8 NF-014 — Potential message inconsistency between serialization and deserialization

During fuzz testing, we spotted a potential anomaly in the message handling between `to_armored_bytes()` and `from_armor_single()`.

Exporting (serializing) and then immediately re-importing (deserializing) a single message may fail in some cases. Due to time limitations, we have not looked into the details, but expect this to be related to the logical differences between the library interfaces for handling inputs containing "one" or "many" messages, and creating multiple messages on export.

```
#[test]
fn message_roundtrip_valid_to_armor_error_from_armor1() {
    let bad_input: &[u8] = &[132, 1, 0, 167];

    // deserialization from_bytes() of a single message succeeds
    let message_deserialized = Message::from_bytes(bad_input).unwrap();

    // serialization succeeds
    let message_armored_serialized = message_deserialized.to_armored_bytes(None.into()).unwrap();

    // deserialization from single fails (since there are multiple messages ?)
    // "unexpected packet SymEncryptedData"
    let _ =
        Message::from_armor_single(&message_armored_serialized[..]).expect("serialized round trip");

    // alternative code:
    //
    // serialization of multiple messages succeeds (!)
    // let _ = Message::from_armor_many(&message_armored_serialized[..]).expect("serialized round
 trip");
}
```

6 Future Work

- **Adopt and extend fuzz testing**
  During this evaluation, the use of coverage-guided fuzz testing proved to be very effective in finding security-related bugs in the analyzed code base. We recommend adopting some of the provided fuzzer test definitions, running them regularly via some form of developer pre-release task or automated Continuous Integration (CI) pipeline action, and extending the test definitions to cover more functionality. See also https://github.com/rpgp/rpgp/issues/42 for an existing ticket on this topic.
- **Discuss meaningful default limits on computing resources**
  Two findings relate to missing controls on the use of main memory and CPU processing time when processing or decrypting externally provided inputs. By accepting all input configurations as allowed by relevant specifications without additional limits, there is a denial-of-service risk if they exceed available system resources. We recommend discussing appropriate default limits with other PGP-related projects, especially for the new `Argon2id` related functionality introduced with `RFC9580`.
- **Regular security assessments**
  Security is a process that must be continuously evaluated and improved; this review is just a single snapshot. Regular audits and ongoing improvements are essential to maintain a strong security posture.

7 Conclusion

We discovered 5 High, 1 Elevated, 2 Moderate and 6 Low-severity issues during this penetration test.

Overall, the number of discovered code issues was higher than expected. We primarily attribute this to the general effectiveness of our testing approach, and do not see it as an indicator of substandard code quality.

Coverage-guided fuzz testing is a powerful strategy to exercise edge cases in complex processing logic, as is the case for the parsing of different file formats and binary representations of OpenPGP objects in `rPGP`. To our knowledge, neither the prior `rPGP` development nor external audits made significant use of on fuzz testing, contributing to this overall project outcome.

Through fuzz testing, we found a number of practically reachable errors, violated assertions, and other situations that causes the program to crash, permanently affecting the availability of the host application. In several cases and scenarios, we see the crash conditions as practically reachable for remote attackers through crafted PGP message data with a low attack complexity. For example, the denial of service could happen during the initial parsing of messages and signatures, or when attempting to decrypt crafted messages with a symmetric secret, without the need to obtain correct signatures or secrets for the attacker. In cases where an attacker can keep sending malformed messages to the victim, or where unsuccessfully parsed messages are re-processed automatically after program startup by an application which is using `rPGP`, these issues could have a significant continued impact on the usability of a messaging program for its end users.

Notably, while it does not improve the overall availability properties in case of severe errors, we think the choice of using the Rust programming language (and absence of `unsafe{}` code blocks) has served the `rPGP` project well in this regard. For example, due to Rust's mandatory built-in checks and mitigations, none of the attempted out-of-bounds accesses which were triggered internally by these vulnerabilities succeeded, preventing any integrity or confidentiality impact. We would also like to positively highlight the rPGP team's efforts towards improved hardening of SHA-1 against collisions.

Several discovered findings involve mathematical edge cases in the form of integer underflows, which Rust can detect but not automatically avoid. By specifically looking for this class of bugs, we hope to have caught some unintended code paths or future vulnerability opportunities as well, and recommend searching more actively for these cases.

We spotted several functions which would benefit from more explicit documentation on their security properties. Additionally, in some `rPGP` functions and for some library dependencies, the memory sanitization of sensitive values is not yet implemented or enabled. Improvements in this area can help mitigate the impact of information leaks.

We discussed some software design aspects of the library with the rPGP team. The low-level nature of the library and some of the corresponding security-related implications could be documented more explicitly, for example in the area of key usage, key validity, and key revocation checks.

Finally, we found two cases of memory and CPU resource exhaustion, which are most closely related to difficult aspects of the OpenGPG specifications in `RFC9580` and `RFC4480`. When processing a large incoming message or decrypting a message with excessive Argon2 hashing parameters, a victim can be tricked into allocating more memory than is available on their system, or performing slow computations. This can crash the `rPGP` process or cause secondary problems on the system.

We recommend fixing all of the issues found. For most of the discovered code issues, the rPGP team rapidly developed patches during the engagement period, allowing us to perform short retests of the relevant findings in order to ensure that mitigations are effective. The results of the retests are marked in the individual findings.

We would like to thank the rPGP developers Friedel Ziegelmayer and Heiko Schaefer for their assistance, patch development and expert feedback during this review. Additionally, we would like to thank Delta Chat developer link2xt for his effort and assistance in assessing the potential impact of the discovered findings on the Delta Chat application.

Finally, we want to emphasize that security is a process – this penetration test is just a one-time snapshot. Security posture must be continuously evaluated and improved. Regular audits and ongoing improvements are essential in order to maintain control of your corporate information security. We hope that this pentest report, and the detailed explanations of our findings, will contribute meaningfully towards that end.

Please don't hesitate to let us know if you have any further questions, or need further clarification on anything in this report.

Appendix 1 Testing team

Christian Reitter *(pentester)*	Christian is an IT Security Consultant with experience in the area of software security and security relevant embedded devices. After his M.Sc. in Computer Science, he has worked as a developer and freelance security consultant with a specialization in fuzz testing. Notable published research includes several major vulnerabilities in popular cryptocurrency software and hardware wallets. This includes remote recovery of wallet keys with weak entropy, remote code execution on hardware wallets, remote theft of secret keys from hardware wallets and circumvention of 2FA protection. He also discovered multiple memory issues in well-known smartcard driver stacks.
Melanie Rieback *(approver)*	Melanie Rieback is a former Asst. Prof. of Computer Science from the VU, who is also the co-founder/CEO of Radically Open Security.